OCTOMAP IMPLEMENTATION FOR EFFICIENT SONAR RAY-CASTING IN SLAM *

Henrique Ribeiro, João Lucas, João Costa, Tiago Duarte
Student
FEUP
Porto
{up201806529, up201806436, up201806560, up201806546}@up.pt

ABSTRACT

The goal of SLAM algorithms (Simultaneous Localization and Mapping) is to map an environment navigated by an autonomous vehicle, while simultaneously locating it in the map, without access to pre-existing charts or external devices. This paper will detail the design and implementation of a system that focuses only on the mapping part of SLAM, developed in the context of the C.U. EDAA. This includes, describing the chosen algorithms and data structures, with a highlight on the developed octree, the probabilistic mapping approach and chosen sonar data filtering methods.

***K*eywords** SLAM, octomap, sonar, ray-cast, dda, bresenham, computer vision

1 Introduction

The development of this project was proposed by the CRAS lab at FEUP, presenting a problem of mapping the bottom of the ocean using a *SHAD AUV*. This *AUV* collects information of its environment using a sonar.
The *AUV* will navigate underwater environments, which presents some challenges that are associated with this environment, such as: the lack of wireless communication between the *AUV* and the main land; the diverse types of noise in the collected data; the limited computation capabilities of the device. On top of that, the characteristics of the problem bring problems related to time constraints in real-time systems: the scans need to be analysed in time for the vehicle to decide how/where to move.

2 Project Definition

For the first part of the project, the main goal is to focus on the 2D mapping of the environment. To achieve this, the main data structure (Octomap) and some image processing methods must be implemented.

2.1 Sonar Data

The CRAS lab at FEUP performed the measurement of all sonar data used during the development of this project, using a *SHAD AUV* with a *Tritech Micron SONAR*[1], visible in Fig 1. In all measurements, the vehicle was inside a squared water tank alongside a floater.

The datasets contain several sequential scans, where groups of beams compose each scan. A scan has beams that span in a 360^a radius around the *AUV*.
Each beam is sent at an angle, and the area that it covers can be seen as a slice of a sphere, as can be seen in Fig 2. In addition, all beams are split into several bins across an interval, and each bin has its respective measured intensity.

A polar image can be created easily from a scan, where each nth row of the image is the nth beam of the scan, and each nth column of the image corresponds to the nth bin of all beams. A polar image can be converted to Cartesian space

* *Citation*: **Henrique R. et al. Octomap implementation for efficient sonar ray-casting in SLAM.**

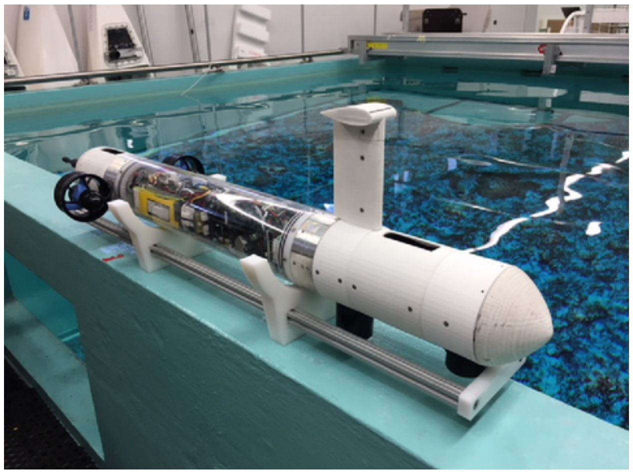
Figure 1: CRAS' SHAD AUV

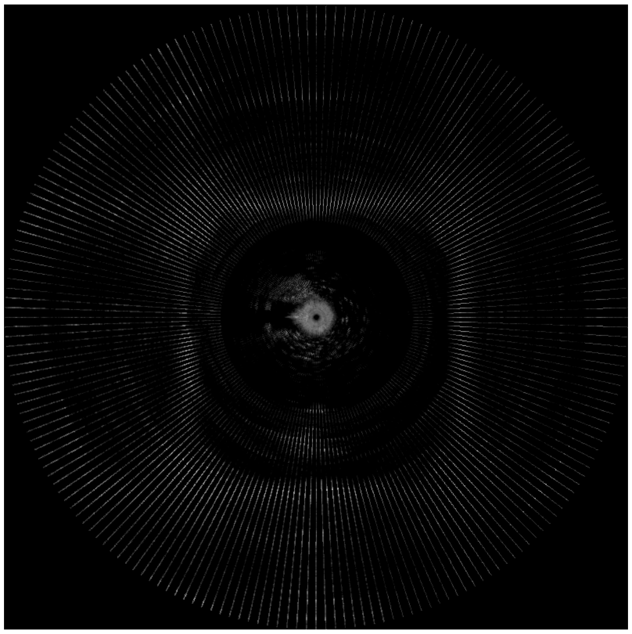
Figure 2: Scan's Cartesian Representation Without Interpolations

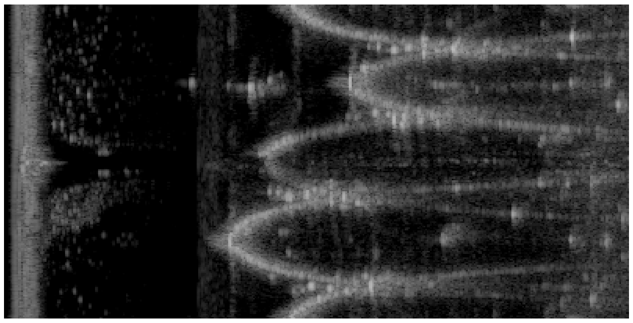
Figure 3: Scan's Polar Representation

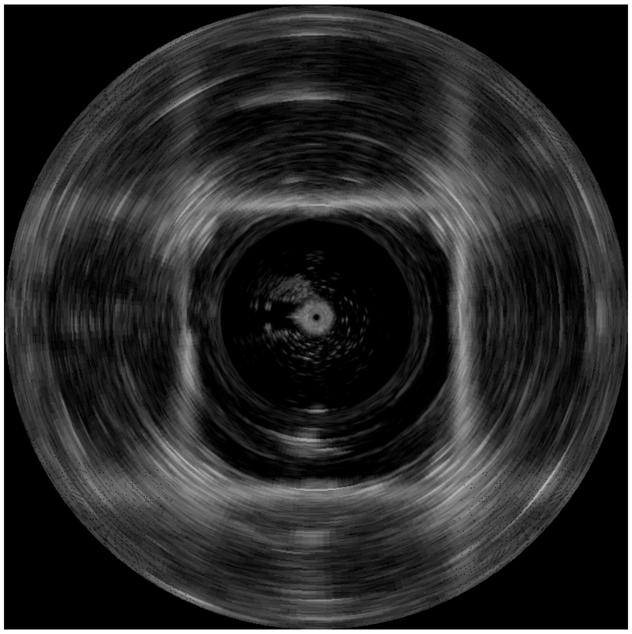
Figure 4: Scan's Cartesian Representation

using angle interpolation. The polar representation can be found in 3 and the Cartesian representation can be found in Fig 4.

3 Mapping Representation

When idealizing the project, three data structures were discussed: occupancy grids, feature based maps and octomaps.

3.1 Occupancy Grid

An occupancy grid maps the environment into a binary matrix (2D or 3D). This can be seen by Fig 5. Usually, zeros represent unknown/occupied locations, and ones represent free cells. This data structure is easy to implement and represents the state directly. Even though this seems a good approach to the problem, it's fixed map size does not allow for it to scale and since everything needs to be instantiated at the start, it is very spatially inefficient. On top of that, in occupancy grids, detail representation is limited.

3.2 Feature Based Map

A feature based map, as can be seen by Fig 6, maps an area through points of interest, using unique features to help locate the vehicle. But it also suffers in some aspects, such as finding the features can be very hard and need a precise algorithm to do so and it's difficult to store the obtained info efficiently

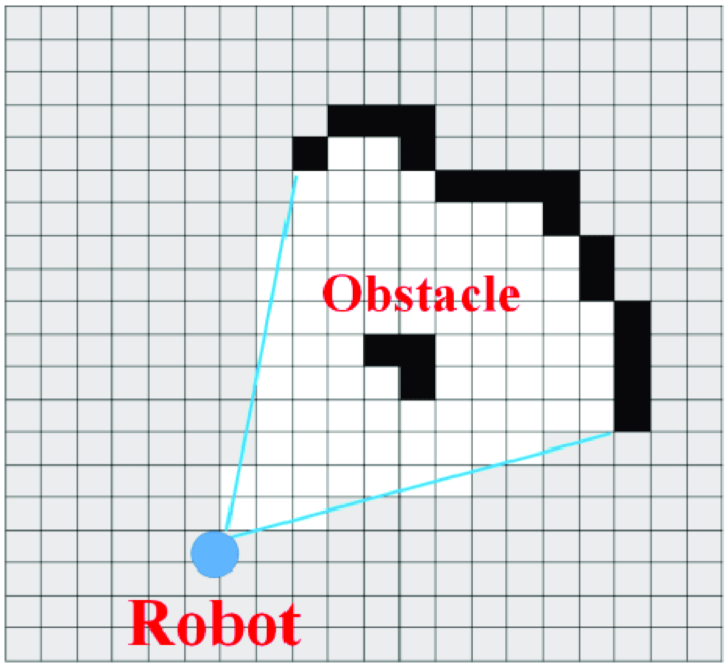

Figure 5: Occupancy Grid Map Representation

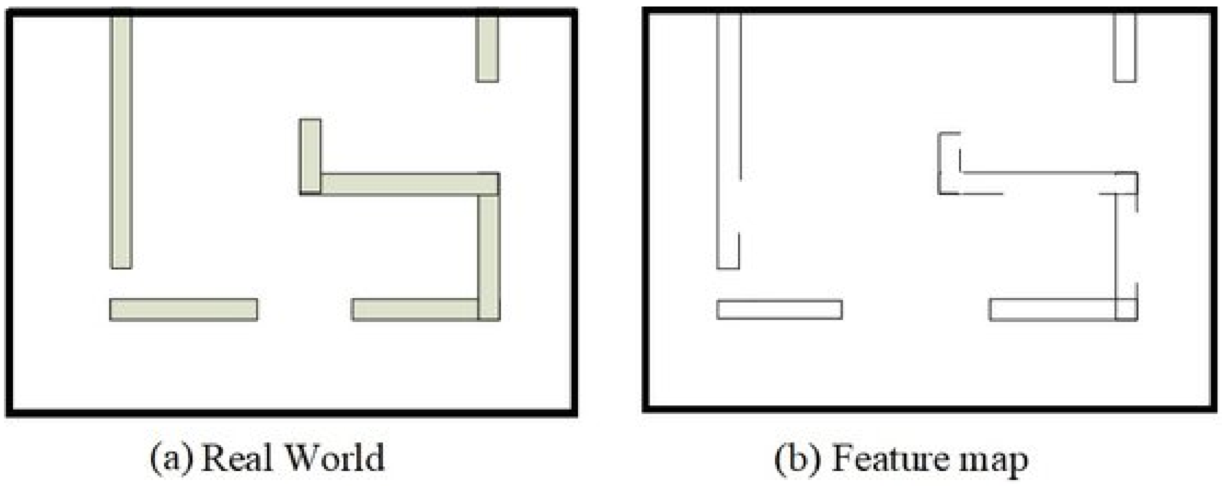

Figure 6: Feature Based Map representation

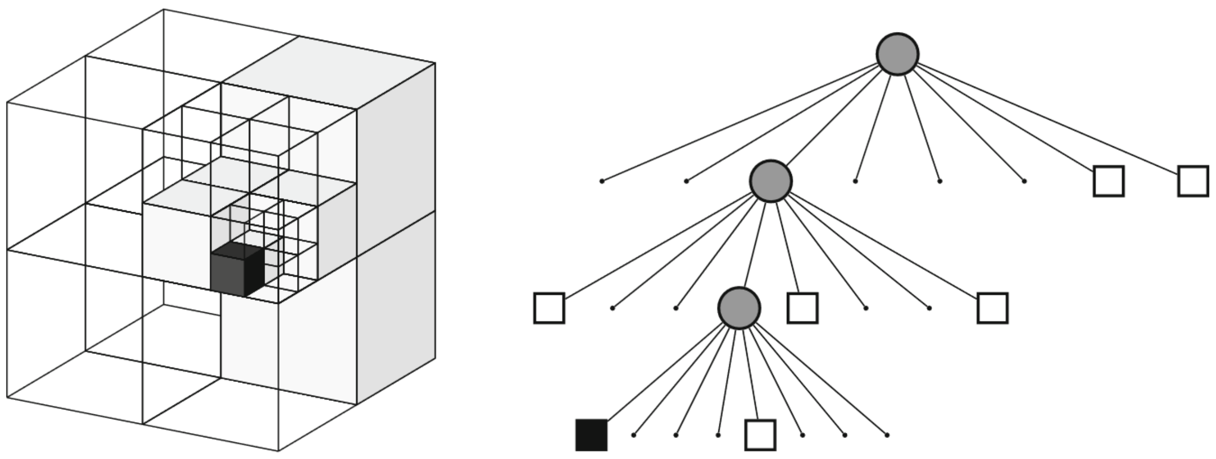
Figure 7: Octree representation (Hornung et al.)

3.3 Octree

An Octree is a spacial data structure that maps a 3D environment. As can be seen in Fig 7, each node represents a space in the 3D environment, called a voxel. This volume can be subdivided into 8 new voxels recursively. This can happen until a minimum voxel size is reached. The minimum voxel size determines each node's resolution. Since the octree is a hierarchical data structure, this means an octree can be cut at a specific depth in order to obtain a more compact representation of the environment. Each node of the octree contains a value, represented by a log odds value, that indicates the certainty that the corresponding node is either empty or filled.

4 Probabilistic Mapping

Since the goal of this project is to map underwater environments using a sonar, some measurements can be afflicted with some uncertainties because of refraction and other errors associated with light travelling underwater. Using a probabilistic measure, we can mitigate these uncertainties by using multiple uncertain measures, which can create a more accurate estimate.
The probability calculation equation, derived from Bayes' Theorem, can be seen bellow 1. It takes into account previous measurements, $z_{1:t-1}$, and the current measurement, z_t.

$$P(n|z_{1:t}) = \left[1 + \frac{1-P(n|z_t)}{P(n|z_t)} \frac{1-P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1-P(n)}\right]^{-1} \tag{1}$$

5 Octomap

To achieve the project's goal, it was decided that an Octomap would be used to store the map information. This implementation represents 3D space using an octree as its main data structure.

The group based this implementation of the Octomap on the OctoMap library and respective technical paper [2] improving and tailoring it to the domain case.

5.1 Node keys

Tree nodes do not store their location information (only their log-odds value of occupancy, as discussed in the log-odds subsection). This saves a very significant amount of space: 3 floating point value per node.

Keys identify nodes. The conversion between coordinates and keys is lossy: keys are only as precise as the tree's depth and resolution. To obtain a key, we take each coordinate (x, y, z) and multiply it by the *resolution factor*, which is $1/tree\_resolution$. The results of these conversions are each stored in an integer.

The integer used needs to have at least as many bits as the maximum tree depth. This is because, during the tree traversal (using the key), each bit of the trio of values is used once to decide the next node to explore. I.e., the 0th bit of each key value decides the first node to explore (excluding the root node), the 1st bit of each key decides the second node to explore, etc…

The max depth of Octree was set to 16. This means that a key is a group of 3 integers with at least 16 bits. In our case, we used the *uint16_t* type, which is an unsigned data type with 16 bits. By using this type, a key is 48 bits, which is the biggest key we can have without exceeding 64 bits. For ray casting purposes (discussed in the ray-casts section), some methods make use of unordered sets of these keys. By keeping the keys smaller than 64 bits, they are smaller than a pointer. This way, we can store them alongside their hashes in the set without compromising its performance: small table that allows for less cache misses, and no pointers to follow for object access and collision treatment.

It should be noted that, although the OctoMap library [2] also makes use of keys similar to the ones described in this section, they can't support trees deeper than 16 levels. This happens, because their implementation uses (exclusively) 16-bit wide integers for these. Our implementation uses the configured depth of the tree to decide the minimum integer width to use for the given depth. In extreme cases (depths higher than 64 levels), the keys used are based on C++ bitsets [3] (slower) instead of integers, thus allowing for arbitrary tree depths.

5.2 Log-odds

The log-odds function, also called **logit**, is a quantile function associated with the standard logistic distribution.

$$\log \frac{P(occupied)}{1 - P(occupied)} \tag{2}$$

Using this formula, the obtained log-odds value belongs to the interval $[-\infty, \infty]$. Within this interval, the value 0 represents the equivalent of 0.5 in regular probabilities. The higher the value, the higher is the likelihood of the space mapped by that node being occupied.

Finally, there was also implemented an upper and lower threshold for the log-odds values. When updating the log-odds probability, the node can be considered occupied if it's log-odd value is higher than the upper threshold, free if it's lower than the lower threshold otherwise the node is considered unknown.

5.3 Updating intermediate nodes

Whenever a leaf node is updated, the tree may require an update to ensure the intermediate nodes' log odds value remain valid. However, propagating these changes is not always necessary, for example, in the following two scenarios:

- **The leaf is already stable**, as updates are meaningless after a node is marked as stable.
- **The update has a log odds value of 0**, meaning the leaf's value will not change and the remainder of the tree will remain as is.

By performing a search to validate if the update is relevant, we may be able to mitigate unnecessary operations.

To test the impact of this check, we used a normal distribution to create point clouds with varying density. All datasets had a mean of $\mu = 10000$ and varying standard deviations. The low density had $\sigma = 5000s$, the medium density had $\sigma = 50$ and the high density had $\sigma = 5$. The control dataset (with no search) does not change with the density, so any dataset can be chosen.

As can be seen in Fig 8, with a sparse point cloud, the performance with the addition of the check was worse than the control. This can be explained due to the overhead introduced not outweighing the time saved.

While the sample with medium density performed slightly better than the control, the most dense dataset performed significantly better with the new checks in place. As the points become denser, the larger the performance boost. As the data received from the sonar is more densely packed, we can safely assume this check will be very useful to the project.

5.3.1 Update strategy

To decide upon the value of the intermediate nodes, we can adopt multiple strategies, such as the **mean** or **max** value of the children.

Using the mean minimizes the error of the intermediate values and represents a more optimistic approach. However, we want to prioritize the *AUV* not hitting any obstacles, so a more conservative approach may be more indicated.

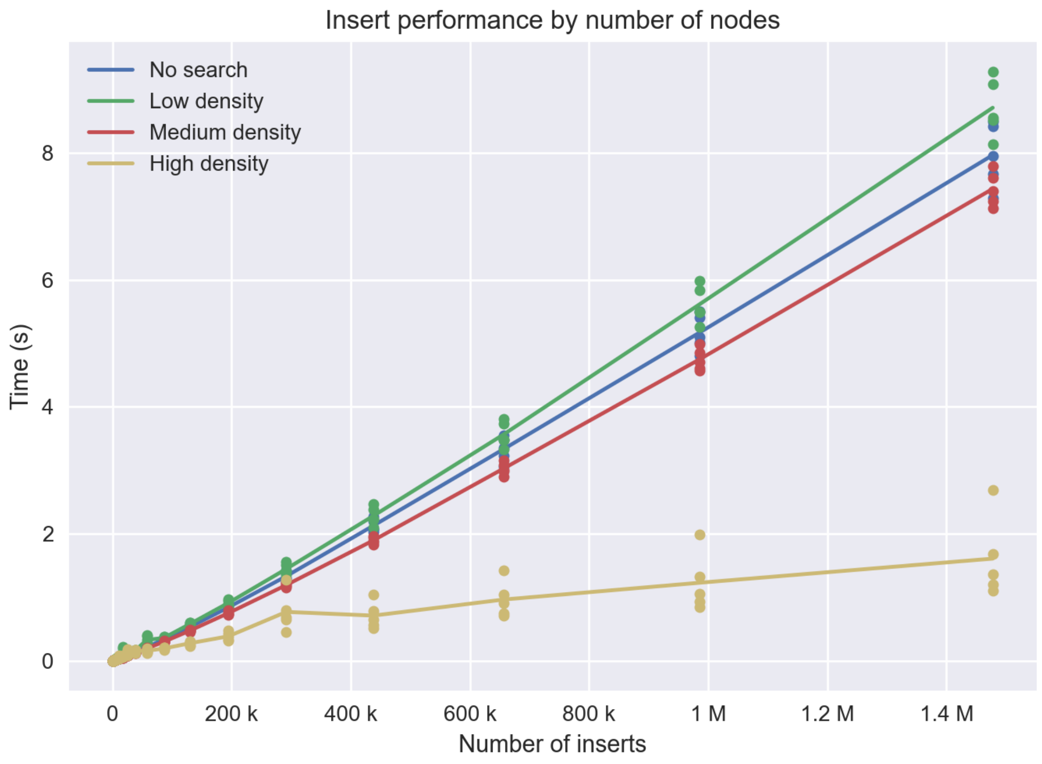

Figure 8: Insert performance chart

	Wo/ Pruning	W/ Pruning	Lazy Eval
Number of Nodes	1 145 977	19 769	19 769
Avg execution time of 5 runs	5.31 s	13.8 s	2.9 s
Number of Intermediate Nodes	145 977	5 201	5 201
Number of Leaf Nodes	1 000 000	19 769	19 769

Table 1: Performance of the pruning operation

The max simply assumes the largest log odds of a child being occupied and assumes its value. This pessimistic approach is more likely to apply successfully to the SLAM problem.

5.4 Pruning

As we go deeper into the data structure, the likelihood of all subdivisions of a voxel agreeing on a given value is higher, for example, all children becoming stable. Instead of keeping all of those leaf nodes, we may be able to merge them into the higher level, thus reducing the space required by the octomap.

To implement this feature, we added a pruning operation to every update. Whenever a leaf is updated, we check if all its sibling nodes have the same value. If they agree on the same log odds, we prune all those leaf nodes and set their parent as a new leaf node.

This check is recursive, so every time a node is pruned, we check if it could be pruned any further.

Table 1 presents the results of inserting the data relative to a $100cm \times 100cm \times 100cm$ cube.

From these experimental results, we can conclude that the number of nodes kept in memory is significantly lower. This comes at a high cost in computation time. To address this issue, we implemented lazy evaluation.

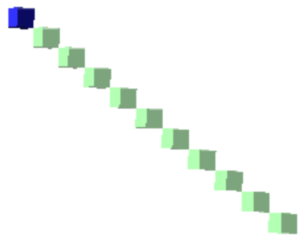

Figure 9: Diagonal Bresenham ray

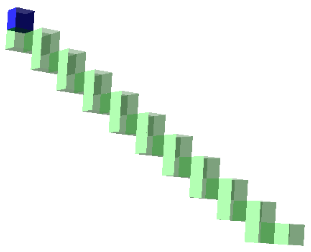

Figure 10: Diagonal DDA ray

5.5 Lazy evaluation

When a batch of updates is required, pruning the tree and updating the respective intermediate nodes at every update represents a significant portion of the processing time spent. This was confirmed by the values obtained in the 1 test.

The solution for these batch operations is to leave the pruning and intermediate node corrections until after all updates have concluded. This was achieved through the lazy evaluation, which, by default, delays all pruning and updates until a synchronize method is called. Until this function is called, there is no guarantee the intermediate log odds values are accurate.

Instead of updating the nodes as we travel back up the tree after an update, we iterate through the entire tree only once to update all values and prune the required branches.

Through this lazy evaluation, we managed to keep the advantages of pruning whilst still reducing the original execution time. As can be seen in table 1, the execution time diminished by 2.4 seconds, with the reduced number of nodes.

5.6 Saving space

Following Hornung's approach[2] instead of keeping 8 pointers to children, we keep a pointer to a single array of 8 children. As such, the leaf nodes only have a single null pointer instead of eight.

Assuming a 64-bit architecture and the dense cube example, where we had 19 769 leaf nodes after the pruning, the implementation of this optimization diminished the size of the data structure from 1.26 MB to 158 KB.

6 Ray-casts

Ray-casts are useful to incorporate the information of each beam into the map. To do this, the scans (and its beams) are processed in order to find the location of the first object hit by each beam (discussed in the edge detection section). By casting a ray from the *AUV* to the location of the first hit of each beam, we obtain the cells that the beam passed through. As we are primarily concerned with mapping the empty space, we want to update these cells (the cells the beam passed through) as empty, and the final cell (hit cell) as occupied with a confidence based on the intensity.

Whilst the traditional application of ray casting techniques intend to find the first obstacle found by a beam fired in a certain direction, we already know where the beam hit. As such, we are interested in knowing which voxels the sonar signal traversed until it hit the obstacle.

In our research, we came across and implemented two solutions, Bresenham's line algorithm (example show in figure 9), and Digital Differential Analyser (DDA) (example show in figure 10).

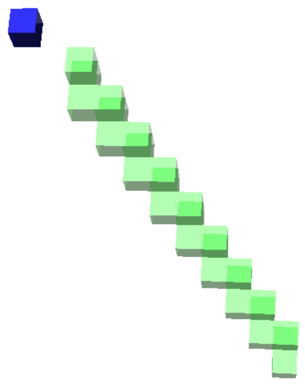

Figure 11: DDA ray missing end

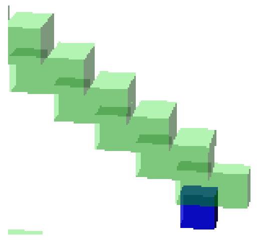

Figure 12: DDA ray snapping at the end

6.1 DDA

Our implementation of the DDA algorithm follows the recommendations of John Amanatides and Andrew Woo in "A Fast Voxel Traversal Algorithm for Ray Tracing" [4] with slight adaptations.

In that paper, the algorithm has 2 phases: the initialization phase and the incremental phase. In the initialization phase, we determine the origin voxel (cell where ray start), the direction of the variation of the coordinates (1 or -1 for (x, y, z) based on whether the coordinate grows or decrements during the ray, respectively), and the value of tx, ty, and tz at which the ray first crosses the respective boundary. In contrast to what is described in the paper, in our approach, these values of t are determined using the tree resolution (the paper assumes the voxel size is 1). These t values are used to determine the $tDelta$ values: how far along the ray must move (in units of t) to cross a voxel. In the incremental phase, we select the smallest of the 3 t values and increment it by the respective $tDelta$ value. Starting from the coordinates of the origin voxel, the next voxel coordinates are found by incrementing the coordinate corresponding to the dimension selected by t. This phase ends when we reach the end voxel. Because of discretization errors of floating-point values (in the increment of the t values), this end-condition isn't always reachable. To handle this case, it is common to stop iterating when all t values are greater than the distance from the origin to the end of the ray, but this can lead to missing voxels in the ray, as shown in figure 11.

To handle this last case, we calculate the travelled distance since the origin to the current voxel and compare it with the distance from the origin to the end voxel. Even if we haven't reached the end voxel, we stop iterating when we exceed this distance. This handles the problematic cases, but sometimes causes a slight *snapping* at the end of the rays, as shown in figure 12.

It should be noted that, because the nodes' keys are related to the coordinates, we use the keys' components to iterate to the next voxel (increment (x, y, z)) instead of floating point coordinates.

6.2 Bresenham's line algorithm

In contrast to DDA, Bresenham's is a much simpler and efficient algorithm. This algorithm is based in integer calculations, which are more efficient and eliminate the discretization errors mentioned in the previous chapter.

The algorithm start by finding the step, the direction of the variation of each coordinate (1 if it grows along the ray, -1 otherwise), and the main dimension. The main dimension is the coordinate of the highest variance. Afterwards, the

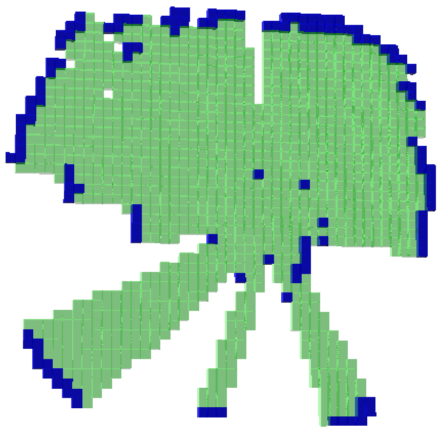
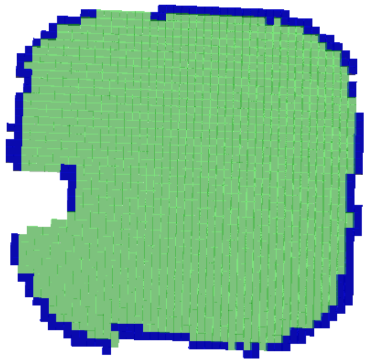

Figure 13: Mapping without image smoothing. Figure 14: Mapping with Gaussian smoothing.

algorithm iterates starting from the origin coordinate by summing the step to the main dimension, and accumulating error in the other dimensions. Whenever the accumulated error in a dimension is high enough, we also step in that direction (resetting the error). It should be noted that, in contrast to the DDA algorithm, this algorithm never misses the end/target voxel, but can create a perfect diagonal ray, which does not resemble a ray from the *SHAD AUV*.

With this, it is possible for all dimensions to be updated at the same time (same iteration) which results in the ability for the ray to travel diagonally. This isn't possible with the DDA algorithm. For this reason, Bresenham's rays are a lot more similar to *laser rays*, which isn't a desirable quality in our use-case.

Given the performance benefits and simplicity of the algorithm, the group decided to use this one, despite the more *laser like* approach.

7 Visualization

Octovis is a tool available in [5]. This tool was used during the development of the project to aid with the visualization of the state of the created data structure. To do this, the data structure had to be written to a binary file following the format read by octovis. This format consists of a sequence of bits representing if the node was unknown (00), occupied (01) empty (10) or had children (11)[2].

8 Data Preprocessing

Before any kind of environment mapping can take place, some preprocessing must be done to reduce noise and find relevant obstacles to be used to detect empty space.

8.1 Sonar Image Smoothing

The presence of noise caused by beam reflections can hinder greatly the obstacle detection phase. For instance, Figure 13 showcases the mapping of an environment without any kind of smoothing. As can be seen, some detected obstacles are in fact false positives, caused by misleading intensities, which are, in turn, caused by noise.

To mitigate this, the sonar data will be considered as an image and the noise reduction problem will be treated with image smoothing methods. Converting the image into the Cartesian coordinate system propagates these errors even further, so the image will be processed in polar coordinates.

The vertical stripe on the left side of the polar image16 represents the band of reflections from the body of the AUV itself. These values are always present and easily predictable, so they can easily be filtered out from the data and provide a cleaner image.

The group experimented with the mean, median and Gaussian filters, which are compared side by side in image 15. The initial exploration was done in Cartesian coordinates, but the results obtained were also applicable in the polar coordinate

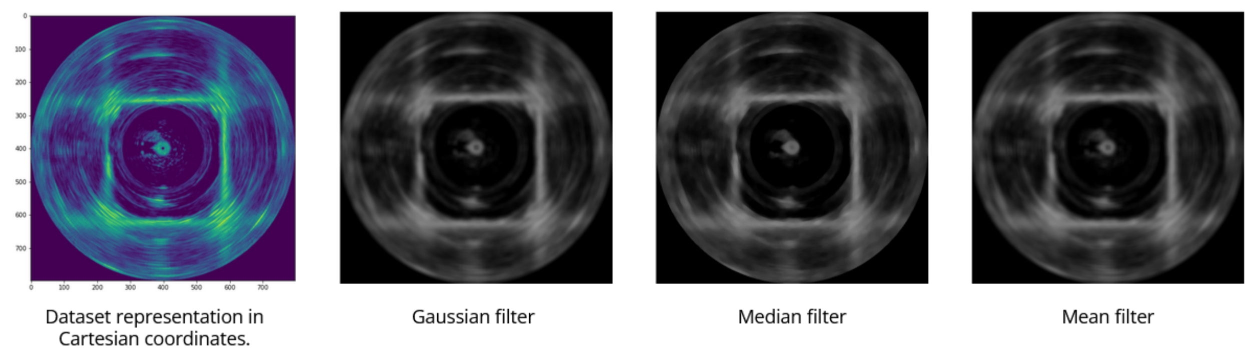

Figure 15: Comparison of different filters

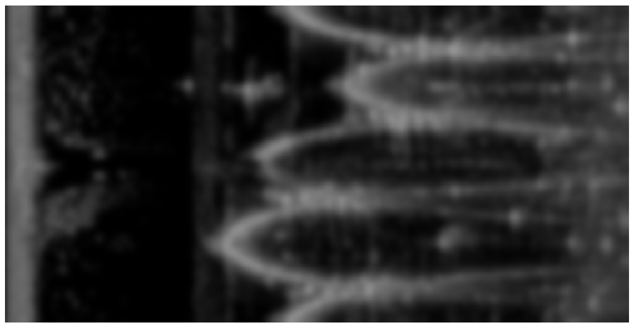

Figure 16: Gaussian filter in polar coordinates

system, as can be seen in Figure 16. As expected, the mean filter is highly sensible to outlier noise, and as such, produces worse results for the problem at hand. Both median and Gaussian filters formed satisfactory smoothed images, but after tweaking the Gaussian parameters, it revealed to handle smaller amounts of noise better when compared with the median filter. For this reason, the Gaussian method was used for most of the examples presented in this paper. Using smoothing clearly improves results, as can be seen in Figure 14.

8.2 Edge Detection

Detecting obstacles accurately in a beam is an essential process when mapping the environment. Free, occupied, and unknown cells need to be properly identified in order to produce the best mapping results. The subsequent algorithms won't take into account measurements that are close to the sonar, as those are self-reflections that should be ignored.

The most simple approach is to use one threshold [6] to define when the difference between two sequential measures in a beam is significant enough for the area to be considered an edge. The value of the threshold needs to be defined a priori. Several threshold values were tested for the same scan, and can be seen in figures 17-20. The threshold value that produced the best results is around 70, as it is low enough that it doesn't ignore any real obstacles, while still being high enough to ignore noise. This value, however, is highly dependent on the used data and its pre-processing steps.

9 Future work

Regarding edge detection, some testing with the Canny edge detector showed it could be an interesting avenue to reduce the noise. The results21 obtained by first applying a Gaussian filter with $kernelsize = 11$ and $\sigma = 2.3$ and tuning the Canny algorithm with $minthreshold = 0$, $maxthreshold = 113$ and $aperturesize = 3$. Canny's double threshold assumes any value over the upper threshold to be an edge and any value below the lower threshold to not be an edge. Values in-between are only considered edges if they are connected to another edge above the upper threshold. We believe this step of the Canny algorithm can help accurately detect the first objects hit by the sonar and will explore its effectiveness to the SLAM problem.

The next milestones for the project will deal with 3D mapping (currently the map is 2D), and the localization of the AUV, thus achieving simultaneous location and mapping (*SLAM*).

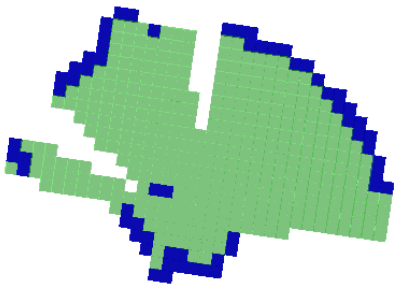

Figure 17: Mapping with 20 thresholds

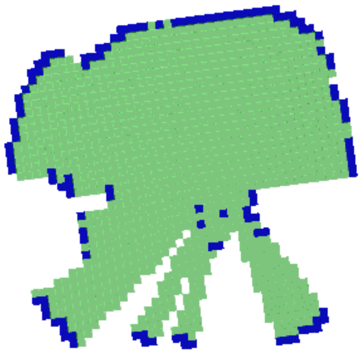

Figure 18: Mapping with 40 threshold

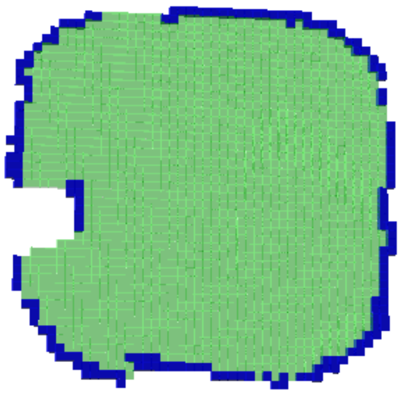

Figure 19: Mapping with 70 threshold

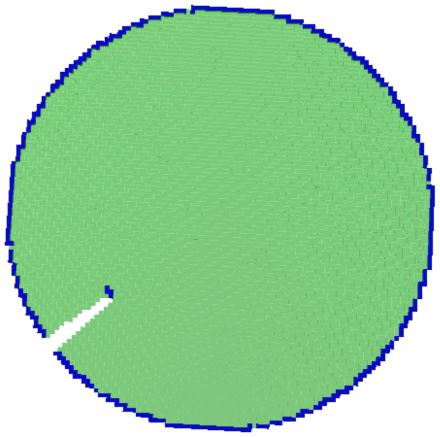

Figure 20: Mapping with 140 thresholds

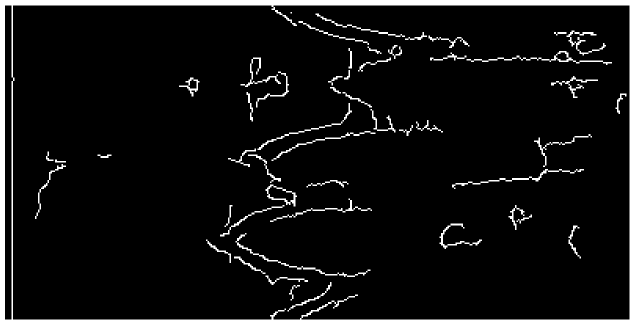

Figure 21: Canny edge detection

10 Conclusion

For this SLAM project, the use of Octomaps proved to be quite efficient. This data structure allowed for significant space optimizations with little temporal overhead. Furthermore, these optimizations end up increasing performance of the more common operations in this data structure.
The two ray-cast algorithms explored, on the other hand, were below expectations. The DDA algorithm sometimes missed the target node, whereas the Bresenham ray-cast algorithm created rays more similar to a laser, which is not the intended result, but it has not yet been problematic due to the low range of the sonar. The chosen edge detection method produced satisfactory results for the data available. This, however, is dependent on parameters that need to be defined a piori, which may not be feasible in real-time underwater mapping.

Acknowledgments

This research was partially supported by Bruno Ferreira and António José from the CRAS laboratory at FEUP. We would also like to thank João Fula for his research paper in the topic, which assisted us in our own.

Once again, we thank António José for his availability to assist us and insights, both from his thesis and from comments/ideas/tips during the development of the project.

References

[1] Tritech. *Micron Sonar Product Manual*. Moog Inc.

[2] et al. Armin Hornung. Octomap: an efficient probabilistic 3d mapping framework based on octrees. *Springer Science*, 2012.

[3] cplusplus.com. bitset - c++ reference. `https://www.cplusplus.com/reference/bitset/bitset/`.

[4] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *EG 1987-Technical Papers*. Eurographics Association, 1987.

[5] Armin Hornung. Octovis. `https://github.com/OctoMap/octomap/tree/devel/octovis`. Accessed on 2022-04-18.

[6] João Pedro Bastos Fula. Underwater mapping using a sonar. *IEEE*, 2020.